Lecture 6: Value Function Approximation

David Silver

overall

lec 3:

Planning (prediction, control) by DP.
Solve a known MDP.

Lec 4

Drop your agent in an unknown MDP with a given policy, how to evaluate this policy, how much rewards we can get if following the behaviors of this policy.

- Model-free prediction
- *Estimate* the value function of an *unknown* MDP

Lec 5

- Model-free control
- *Optimise* the value function of an *unknown* MDP

Find v_*, q_*
We use same tools, we iterate them and find the best possible behaviors.

Lec 6 scale up to real practical problems.

Previous lectures: Use table
Scale Up: Use function approximation.

Lec 6: Function approximation for value based algorithms
Lec 7: Function approximation for policy based algorithms

Outline

Outline

Large-Scale Reinforcement Learning

Reinforcement learning can be used to solve *large* problems, e.g.

- Backgammon: 10^{20} states
- Computer Go: 10^{170} states
- Helicopter: continuous state space

Uncounted infinite states. Cannot build a table anymore.

How can we scale up the model-free methods for *prediction* and *control* from the last two lectures?

Previous methods we discussed:

Use a table, have a separate value for each state.

Scale up:

Table doesn't work (size, efficiency, storage…)

So build a function approximator that estimates the value of the parts of the space you visited, and what generalizes across parts of those space.

For state and its neighbor state, the value should be similar intuitively, we want our function to understand this generalization. So we don't need to store distinct values for those similar states.

How to achieve this generalization? What methods for representing and learning value functions efficiently?

This lecture: function approximation for value based algorithms

Next lecture: function approximation for policy based algorithms

Value Function Approximation

- So far we have represented value function by a *lookup table*
  - Every state s has an entry $V(s)$
  - Or every state-action pair s, a has an entry $Q(s, a)$
- Problem with large MDPs:
  - There are too many states and/or actions to store in memory
  - It is too slow to learn the value of each state individually
- Solution for large MDPs:
  - Estimate value function with *function approximation*

Estimate this mapping

w: weights of features

$$\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$$

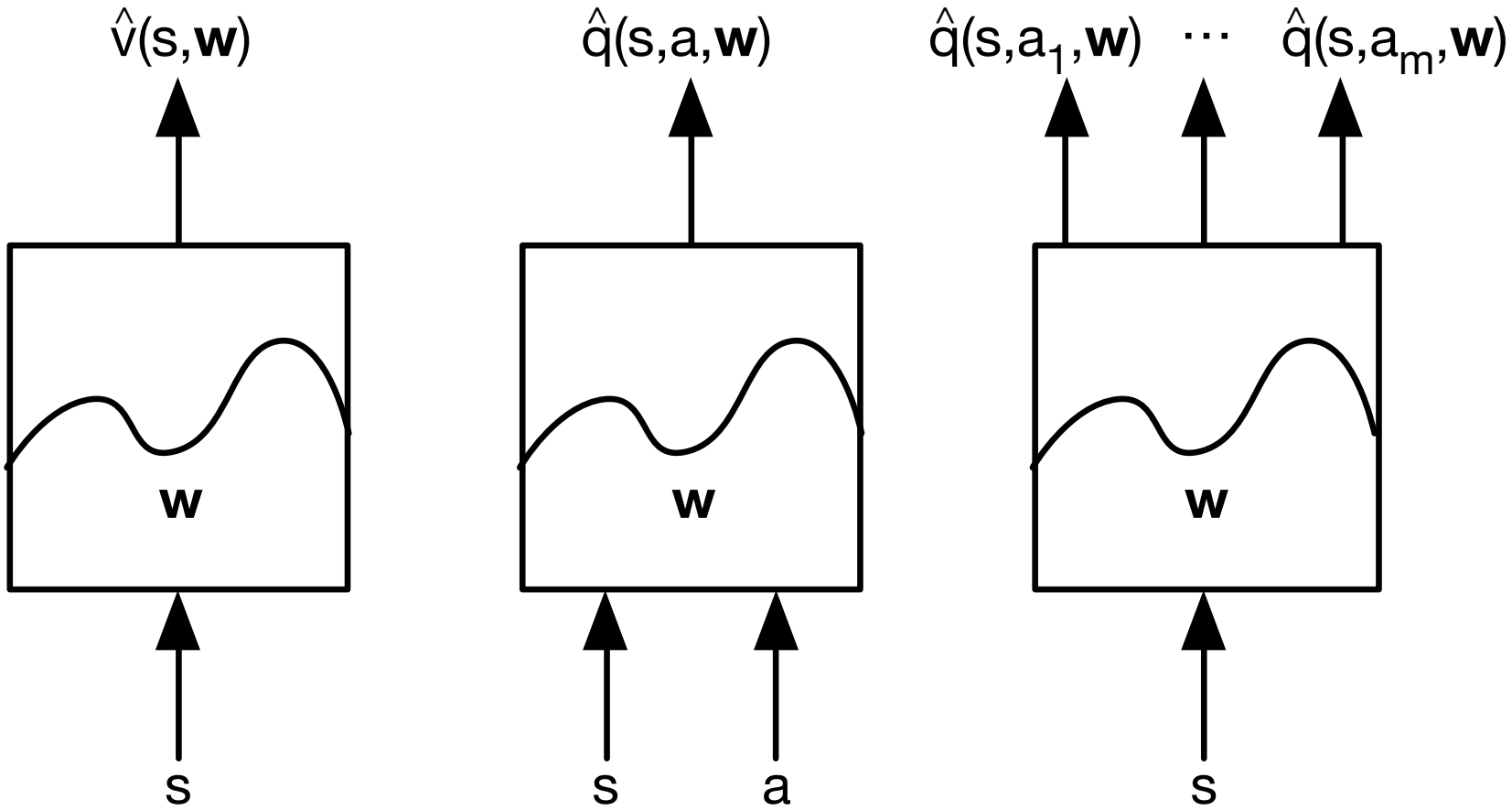
Mapping from state s to true value v_pi

$$\text{or } \hat{q}(s, a, \mathbf{w}) \approx q_\pi(s, a)$$

Pros: 1. reduce memory;
2. allow us to generalize.

  - *Generalise* from seen states to unseen states
  - *Update* parameter $\mathbf{w}$ using MC or TD learning

Types of Value Function Approximation



Which Function Approximator?

There are many function approximators, e.g.

- Linear combinations of features
- Neural network
- Decision tree
- Nearest neighbour
- Fourier / wavelet bases
- ...

Which Function Approximator?

We consider differentiable (可微的) function approximators, e.g.

- Linear combinations of features
- Neural network
- Decision tree
- Nearest neighbour
- Fourier / wavelet bases
- ...

Furthermore, we require a training method that is suitable for non-stationary, non-iid data

Bc our policy also changes

Bc next state and current state are correlated

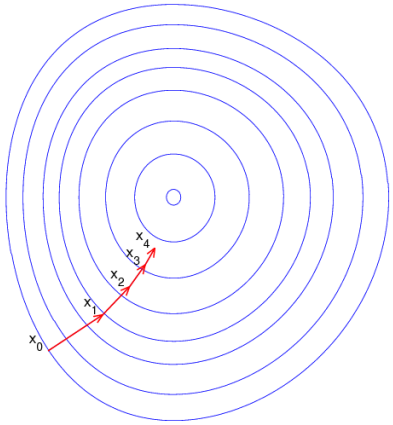
Outline

Gradient Descent

- Let $J(\mathbf{w})$ be a differentiable function of parameter vector $\mathbf{w}$ also is our objective function.
- Define the *gradient* of $J(\mathbf{w})$ to be

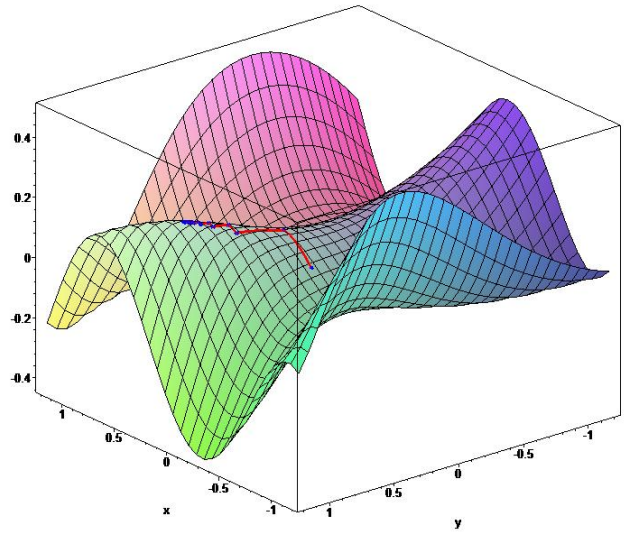
$$\nabla_{\mathbf{w}} J(\mathbf{w}) = \begin{pmatrix} \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}_1} \\ \vdots \\ \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}_n} \end{pmatrix}$$

Gradient of partial derivatives wrt each parameters



This gradient vector tells us about the direction of steepest descent.

- To find a local minimum of $J(\mathbf{w})$
- Adjust $\mathbf{w}$ in direction of -ve gradient



+ve means positive, –ve means negative

Find min of objective function: what we need to do is adjust parameter a little bit downwards step–size*direction of our gradient.

$$\Delta \mathbf{w} = -\frac{1}{2}\alpha \nabla_{\mathbf{w}} J(\mathbf{w})$$

Downwards (opposite of gradient)

Make math neat

gradient

where α is a step-size parameter

Value Function Approx. By Stochastic Gradient Descent

- Goal: find parameter vector $\mathbf{w}$ minimising mean-squared error between approximate value fn $\hat{v}(s, \mathbf{w})$ and true value fn $v_\pi(s)$

$$J(\mathbf{w}) = \mathbb{E}_\pi \left[(v_\pi(S) - \hat{v}(S, \mathbf{w}))^2 \right]$$

true (oracle) – approximate (estimated)

Assuming there is an oracle tell us what the true value is. (This is cheating, we will fix this cheating later)

- Gradient descent finds a local minimum

$$\begin{aligned} \Delta \mathbf{w} &= -\frac{1}{2} \alpha \nabla_{\mathbf{w}} J(\mathbf{w}) \\ &= \alpha \mathbb{E}_\pi \left[(v_\pi(S) - \hat{v}(S, \mathbf{w})) \nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w}) \right] \end{aligned}$$

Error

Gradient of our approximator function

But how to deal with the expectation E?
Use:

- Stochastic gradient descent *samples* the gradient

Instead of doing a full gradient, i.e. calculating the expectation E, we sample a state (randomly) by just seeing which state we visited, we check what oracle says about that state v_pi and our estimation ^v and calc the error and adjust it by gradient.

$$\Delta \mathbf{w} = \alpha (v_\pi(S) - \hat{v}(S, \mathbf{w})) \nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w})$$

Error to correct

Gradient tells you direction to correct

- Expected update is equal to full gradient update

Beautiful thing about SGD is that adjusting online also arrive at min.

Feature Vectors

- Represent state by a *feature vector*

Good feature make learning easier

$$\mathbf{x}(S) = \begin{pmatrix} \mathbf{x}_1(S) \\ \vdots \\ \mathbf{x}_n(S) \end{pmatrix}$$

- For example:
  - Distance of robot from landmarks
  - Trends in the stock market
  - Piece and pawn configurations in chess

How to use features?

Linear Value Function Approximation

One beautiful thing about LCF is we never get stuck in local optimal. We always converge to best correct answer.

- Represent value function by a linear combination of features

$$\hat{v}(S, \mathbf{w}) = \underbrace{\mathbf{x}(S)}^{\text{Feature vector}\top} \underbrace{\mathbf{w}}_{\text{Weights}} = \sum_{j=1}^{n} \mathbf{x}_j(S)\mathbf{w}_j$$

Each feature takes its weight

- Objective function is quadratic in parameters $\mathbf{w}$

Mean square error:

$$J(\mathbf{w}) = \mathbb{E}_\pi \left[(\underbrace{v_\pi(S)}_{\text{True value}} - \underbrace{\mathbf{x}(S)^\top \mathbf{w}}_{\text{Estimated value}})^2 \right]$$

- Stochastic gradient descent converges on *global* optimum
- Update rule is particularly simple

The gradient is just the feature vector

$$\nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w}) = \mathbf{x}(S)$$

$$\underbrace{\Delta \mathbf{w}}_{\text{The changes to weights}} = \alpha(v_\pi(S) - \hat{v}(S, \mathbf{w}))\mathbf{x}(S)$$

Update = *step-size* $\times$ *prediction error* $\times$ *feature value*

Table Lookup Features

- Table lookup is a special case of linear value function approximation
- Using *table lookup features*

It's 1 if in corresponding state, it's 0 if not in that state.

$$\mathbf{x}^{table}(S) = \begin{pmatrix} \mathbf{1}(S = s_1) \\ \vdots \\ \mathbf{1}(S = s_n) \end{pmatrix}$$

- Parameter vector $\mathbf{w}$ gives value of each individual state

$$\hat{v}(S, \mathbf{w}) = \begin{pmatrix} \mathbf{1}(S = s_1) \\ \vdots \\ \mathbf{1}(S = s_n) \end{pmatrix} \cdot \begin{pmatrix} \mathbf{w}_1 \\ \vdots \\ \mathbf{w}_n \end{pmatrix}$$

We just pick out one of these weights bc only one state is 1 all others are 0.

Incremental Prediction Algorithms

We cheat by assuming oracle exists and tell us true value.
But how to solve this cheating in real practice?

- Have assumed true value function $v_\pi(s)$ given by supervisor oracle
- But in RL there is no supervisor, only rewards
- In practice, we substitute a *target* for $v_\pi(s)$
  - For MC, the target is the return G_t

$$\Delta \mathbf{w} = \alpha(G_t - \hat{v}(S_t, \mathbf{w}))\nabla_{\mathbf{w}} \hat{v}(S_t, \mathbf{w})$$

  - For TD(0), the target is the TD target $R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w})$

$$\Delta \mathbf{w} = \alpha(R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w}))\nabla_{\mathbf{w}} \hat{v}(S_t, \mathbf{w})$$

  - For TD(λ), the target is the λ-return G_t^λ

$$\Delta \mathbf{w} = \alpha(G_t^\lambda - \hat{v}(S_t, \mathbf{w}))\nabla_{\mathbf{w}} \hat{v}(S_t, \mathbf{w})$$

Monte-Carlo with Value Function Approximation

- Return G_t is an unbiased, noisy sample of true value $v_\pi(S_t)$
- Can therefore apply supervised learning to "training data":

$$\langle S_1, G_1 \rangle, \langle S_2, G_2 \rangle, ..., \langle S_T, G_T \rangle$$

MC we just roll out to see how much rewards we actually get from each of those states

- For example, using *linear Monte-Carlo policy evaluation*

$$\begin{aligned}\Delta \mathbf{w} &= \alpha(G_t - \hat{v}(S_t, \mathbf{w}))\nabla_{\mathbf{w}} \hat{v}(S_t, \mathbf{w}) \\ &= \alpha(G_t - \hat{v}(S_t, \mathbf{w}))\mathbf{x}(S_t)\end{aligned}$$

- Monte-Carlo evaluation converges to a local optimum
- Even when using non-linear value function approximation

TD Learning with Value Function Approximation

- The TD-target $R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w})$ is a *biased* sample of true value $v_\pi(S_t)$
- Can still apply supervised learning to "training data":

$$\langle S_1, R_2 + \gamma \hat{v}(S_2, \mathbf{w}) \rangle, \langle S_2, R_3 + \gamma \hat{v}(S_3, \mathbf{w}) \rangle, ..., \langle S_{T-1}, R_T \rangle$$

- For example, using *linear TD(0)*

$$\begin{aligned} \Delta \mathbf{w} &= \alpha(R + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})) \nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w}) \\ &= \alpha \delta \mathbf{x}(S) \end{aligned}$$

- Linear TD(0) converges (close) to global optimum

TD(λ) with Value Function Approximation

- The λ-return G_t^λ is also a biased sample of true value $v_\pi(s)$
- Can again apply supervised learning to "training data":

$$\left\langle S_1, G_1^\lambda \right\rangle, \left\langle S_2, G_2^\lambda \right\rangle, ..., \left\langle S_{T-1}, G_{T-1}^\lambda \right\rangle$$

- Forward view linear TD(λ)

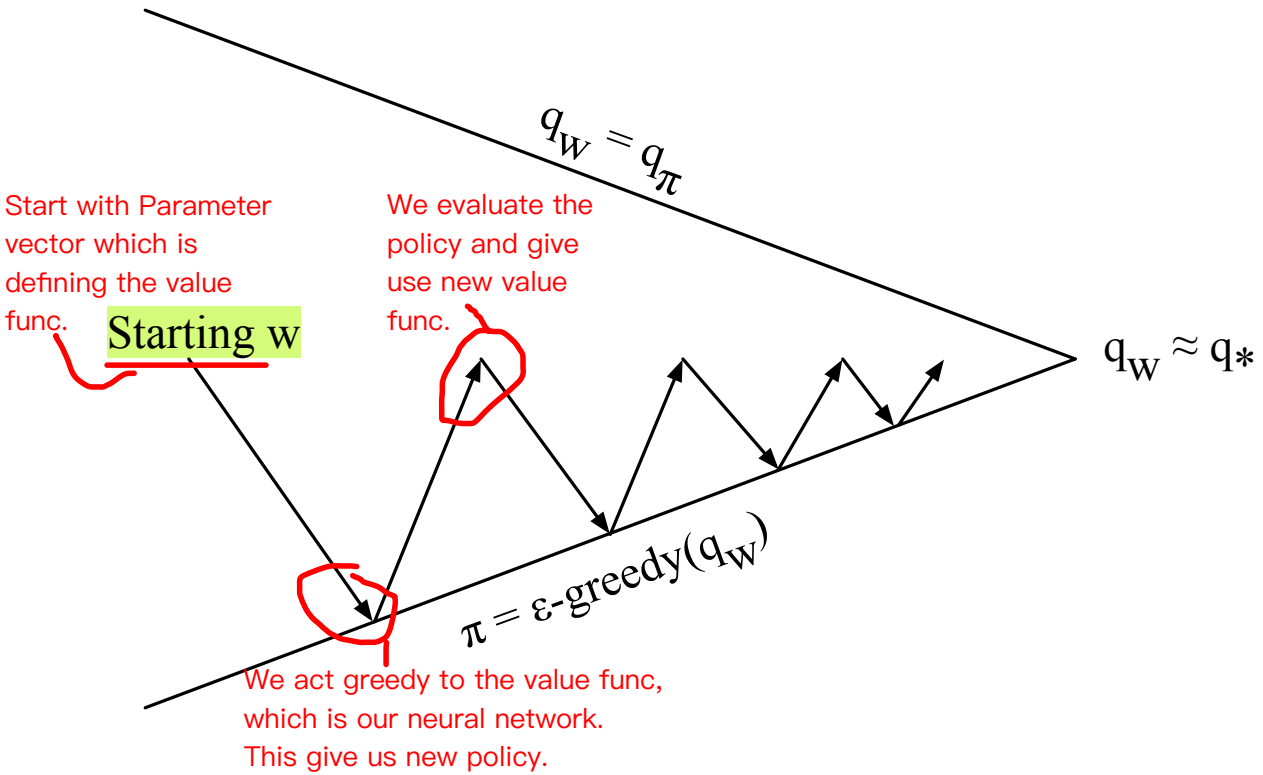
$$\begin{aligned}\Delta \mathbf{w} &= \alpha(G_t^\lambda - \hat{v}(S_t, \mathbf{w}))\nabla_{\mathbf{w}} \hat{v}(S_t, \mathbf{w}) \\ &= \alpha(G_t^\lambda - \hat{v}(S_t, \mathbf{w}))\mathbf{x}(S_t)\end{aligned}$$

- Backward view linear TD(λ)

$$\begin{aligned}\delta_t &= R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w}) \\ E_t &= \gamma\lambda E_{t-1} + \mathbf{x}(S_t) \\ \Delta \mathbf{w} &= \alpha \delta_t E_t\end{aligned}$$

Forward view and backward view linear TD(λ) are equivalent

Control with Value Function Approximation



Policy evaluation Approximate policy evaluation, $\hat{q}(\cdot, \cdot, \mathbf{w}) \approx q_\pi$

Policy improvement ϵ-greedy policy improvement

Action-Value Function Approximation

All the same things using q instead of v

- Approximate the action-value function

$$\hat{q}(S, A, \mathbf{w}) \approx q_\pi(S, A)$$

Predict how much rewards we get from that state and action

- Minimise mean-squared error between approximate action-value fn $\hat{q}(S, A, \mathbf{w})$ and true action-value fn $q_\pi(S, A)$

$$J(\mathbf{w}) = \mathbb{E}_\pi \left[(q_\pi(S, A) - \hat{q}(S, A, \mathbf{w}))^2 \right]$$

- Use stochastic gradient descent to find a local minimum

$$-\frac{1}{2} \nabla_{\mathbf{w}} J(\mathbf{w}) = (q_\pi(S, A) - \hat{q}(S, A, \mathbf{w})) \nabla_{\mathbf{w}} \hat{q}(S, A, \mathbf{w})$$

Error * gradient

$$\Delta \mathbf{w} = \alpha (q_\pi(S, A) - \hat{q}(S, A, \mathbf{w})) \nabla_{\mathbf{w}} \hat{q}(S, A, \mathbf{w})$$

Move a little bit in the direction of the error between the q value I estimated before and the q value oracle told me, multipled by the gradient.

Linear Action-Value Function Approximation

- Represent state *and* action by a *feature vector*

$$\mathbf{x}(S, A) = \begin{pmatrix} \mathbf{x}_1(S, A) \\ \vdots \\ \mathbf{x}_n(S, A) \end{pmatrix}$$

- Represent action-value fn by linear combination of features

Simplest way: linear combination of features
More sophisticated: neural network

$$\hat{q}(S, A, \mathbf{w}) = \mathbf{x}(S, A)^\top \mathbf{w} = \sum_{j=1}^{n} \mathbf{x}_j(S, A)\mathbf{w}_j$$

- Stochastic gradient descent update

$$\nabla_{\mathbf{w}} \hat{q}(S, A, \mathbf{w}) = \mathbf{x}(S, A)$$

$$\Delta\mathbf{w} = \alpha(q_\pi(S, A) - \hat{q}(S, A, \mathbf{w}))\mathbf{x}(S, A)$$

Change the weights in the direction of step–size * error * features

Incremental Control Algorithms

What to do if no oracle?

- Like prediction, we must substitute a *target* for $q_\pi(S, A)$
  - True action–value func (oracle told us)
  - For MC, the target is the return G_t
    - Noisy unbiased estimator of oracle

$$\Delta \mathbf{w} = \alpha(G_t - \hat{q}(S_t, A_t, \mathbf{w}))\nabla_{\mathbf{w}} \hat{q}(S_t, A_t, \mathbf{w})$$

  - For TD(0), the target is the TD target $R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$
    - One step TD target

$$\Delta \mathbf{w} = \alpha(R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}) - \hat{q}(S_t, A_t, \mathbf{w}))\nabla_{\mathbf{w}} \hat{q}(S_t, A_t, \mathbf{w})$$

  - For forward-view TD(λ), target is the action-value λ-return
    - Lambda return

$$\Delta \mathbf{w} = \alpha(q_t^\lambda - \hat{q}(S_t, A_t, \mathbf{w}))\nabla_{\mathbf{w}} \hat{q}(S_t, A_t, \mathbf{w})$$

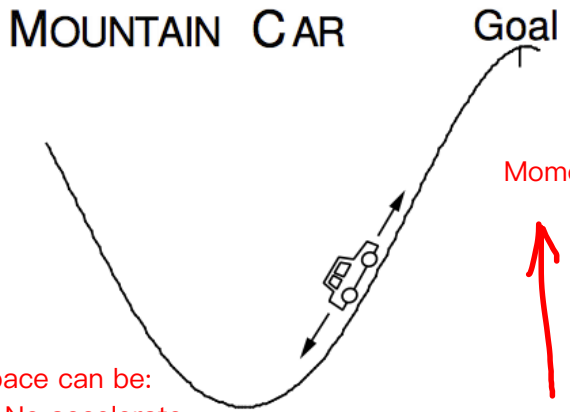
  - For backward-view TD(λ), equivalent update is
    - Eligibility traces

$$\delta_t = R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}) - \hat{q}(S_t, A_t, \mathbf{w})$$
$$E_t = \gamma \lambda E_{t-1} + \nabla_{\mathbf{w}} \hat{q}(S_t, A_t, \mathbf{w})$$
$$\Delta \mathbf{w} = \alpha \delta_t E_t$$

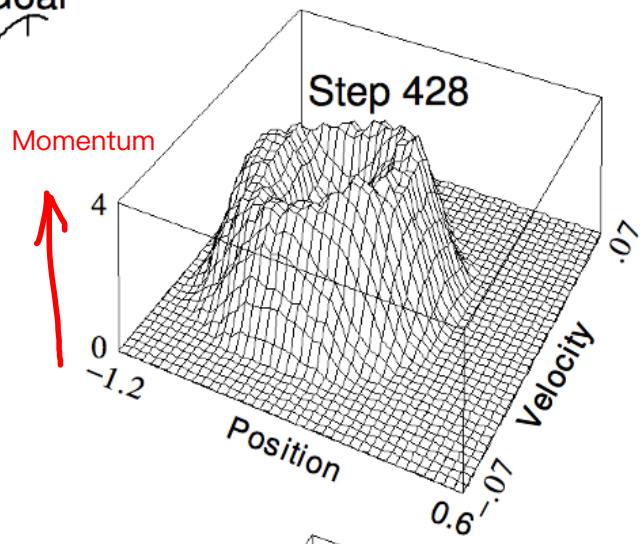
Use q instead of v so we can control. Once has q we can pick best action (we don't need a model to tell us what next action we should take)

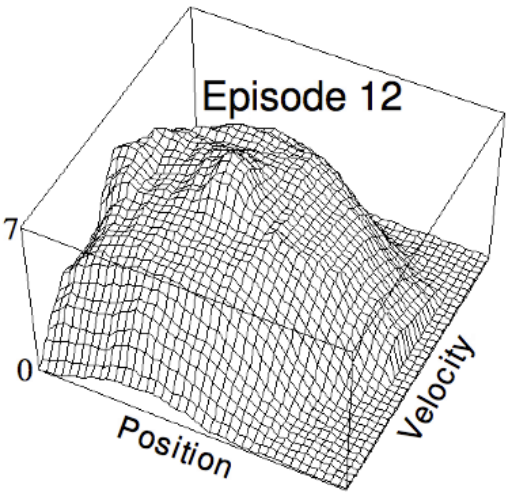
Every single step we update our action–value func using a linear function approximation to estimate q. Every single step we update q and every single step we act greedy wrt q to pick next action and we also flip a coin to see something randomly to make sure we explore. This is Sarsa. The way we update q is using one step td return/target.

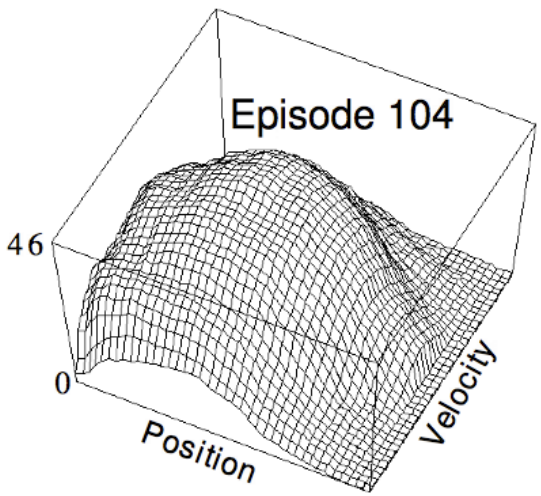
Linear Sarsa with Coarse Coding in Mountain Car

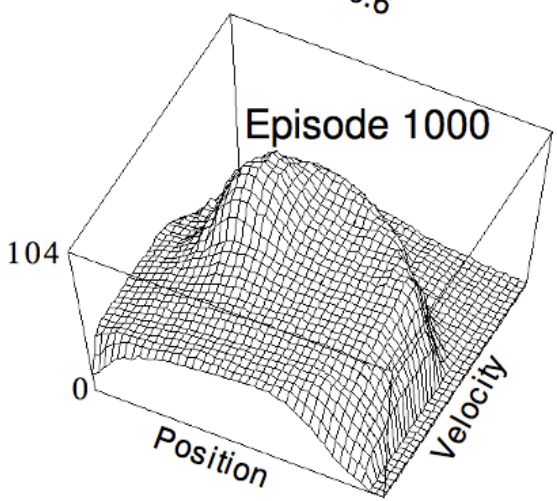


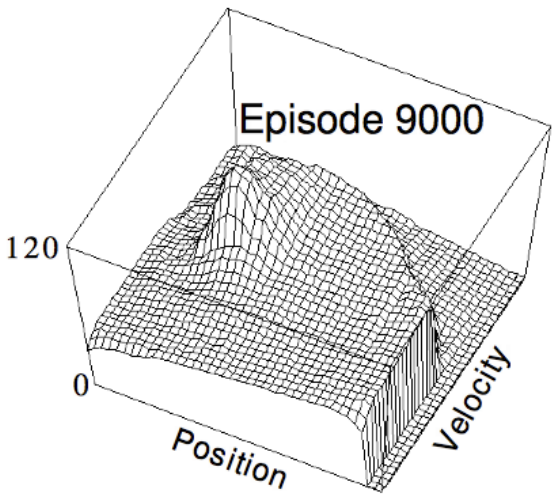
Action space can be:
Option 1: No accelerate
Option 2: Accelerate 1s
Option 3: Accelerate 2s
Option 3: Accelerate 3s



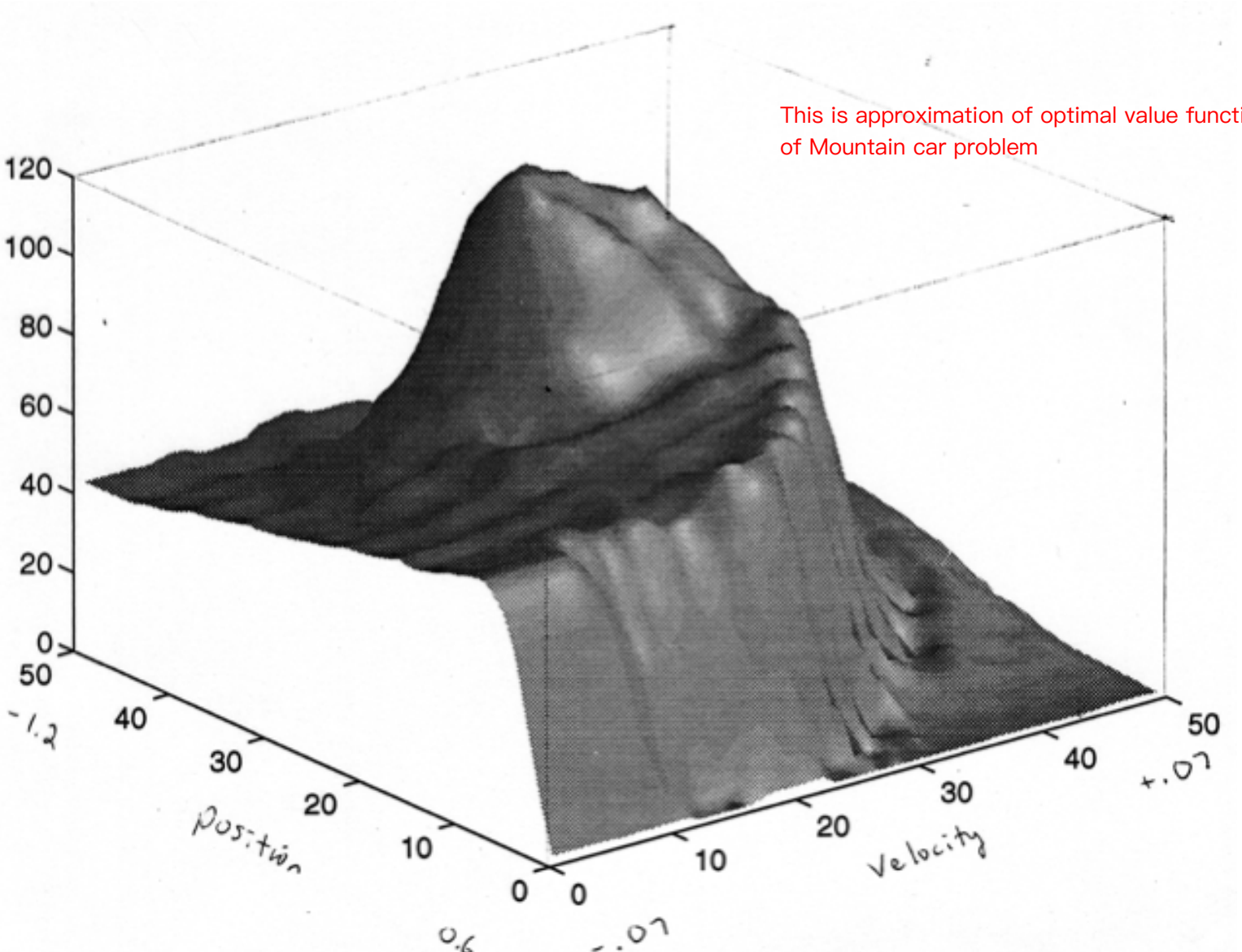




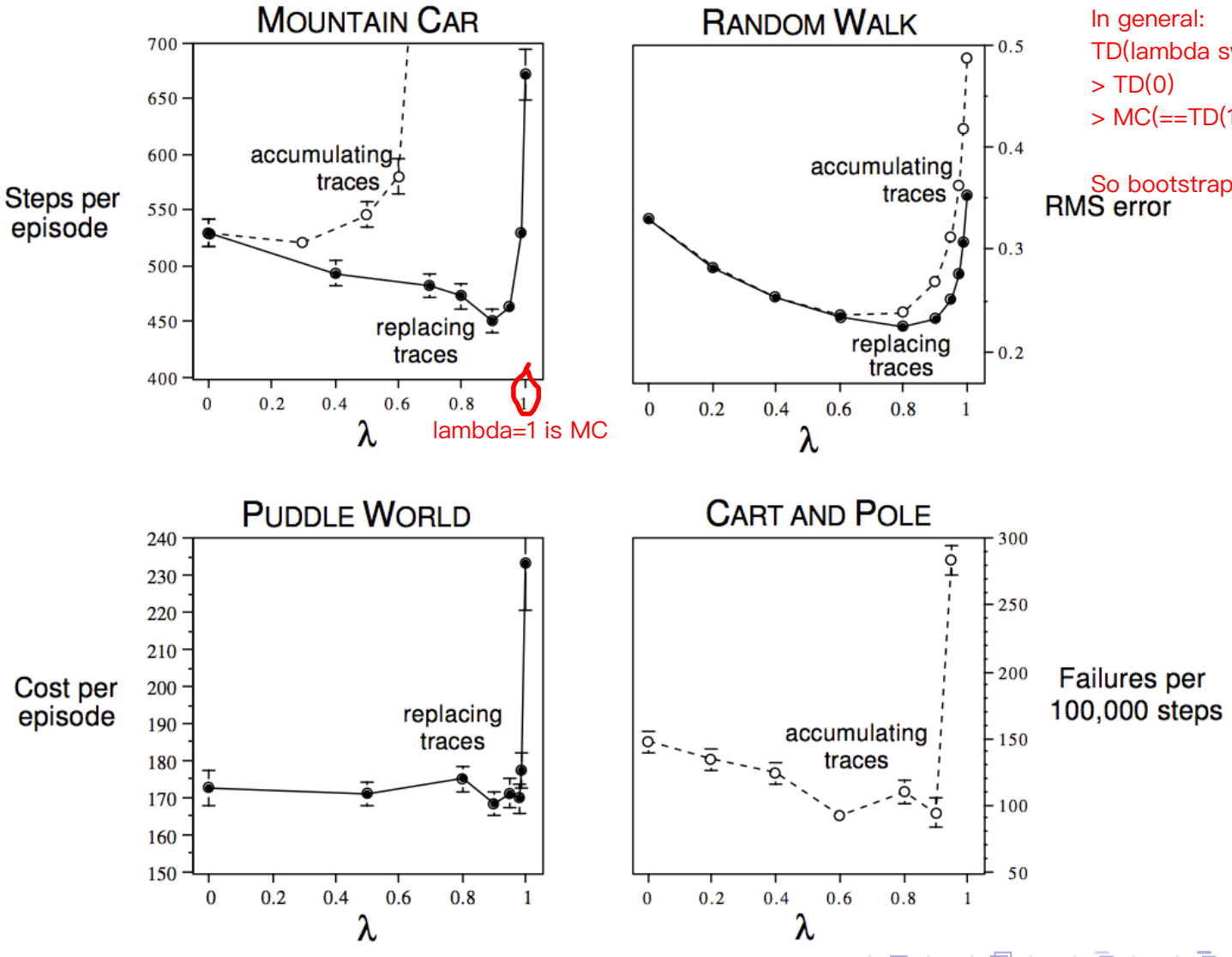




Linear Sarsa with Radial Basis Functions in Mountain Car



Study of λ: Should We Bootstrap?

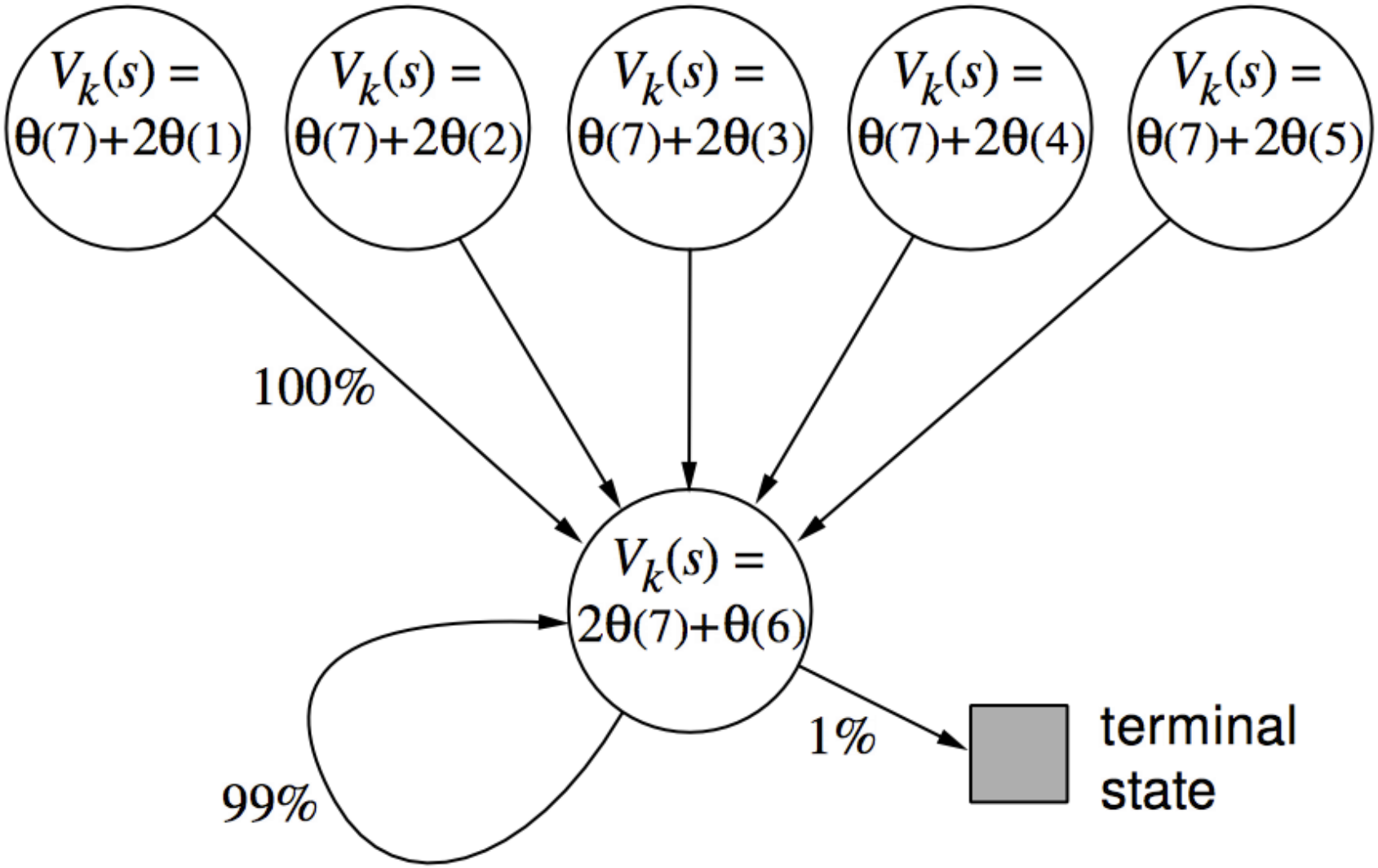


In general:
TD(lambda sweet spot)
> TD(0)
> MC(==TD(1))

So bootstrap helps.

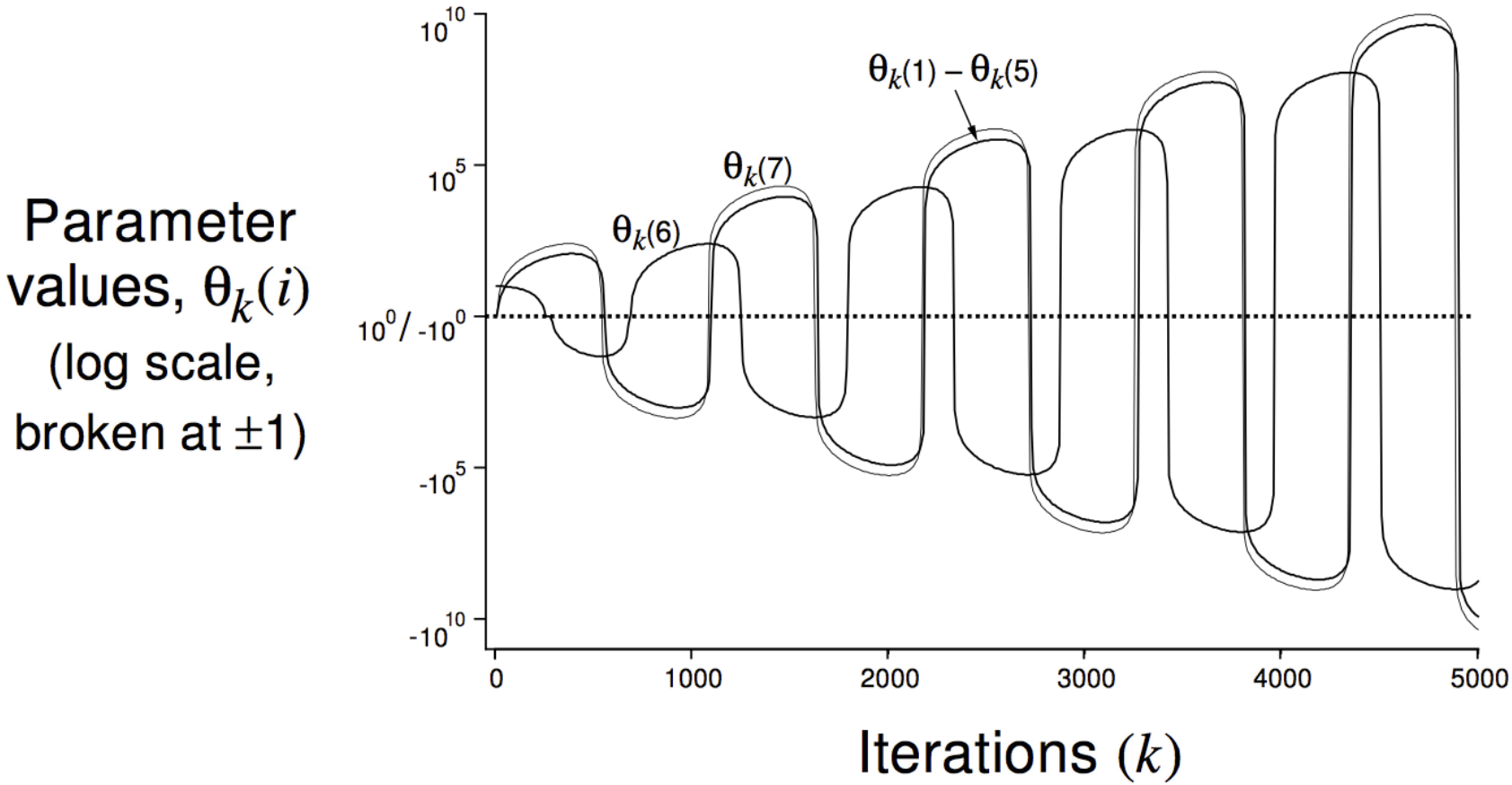
Baird's Counterexample

Counterexample for TD/bootstrap not helps



Parameter Divergence in Baird's Counterexample

Counterexample for TD/bootstrap not helps.
So TD don't guarantee stable. It's important to know when it's better to use TD when not.



Convergence of Prediction Algorithms

Summary for when it's ok to use TD
(when guarantee to converge)

Guarantee to converge to right answer.

Possible to diverge. Don't use TD.

Like neural network (how to address nn will be discussed in part3)

Policy evaluate/Prediction only

On/Off-Policy	Algorithm	Table Lookup	Linear	Non-Linear
On-Policy	MC	✓	✓	✓
	TD(0)	✓	✓	✗
	TD(λ)	✓	✓	✗
Off-Policy	MC	✓	✓	✓
	TD(0)	✓	✗	✗
	TD(λ)	✓	✗	✗

We learn about the policy we follow

Converge means parameter vector (weight vector) getting closer and closer to a fixed value which is the best parameter vector (what we want) for our function approximator.

Gradient Temporal-Difference Learning

- TD does not follow the gradient of *any* objective function
- This is why TD can diverge when off-policy or using non-linear function approximation
- Gradient TD follows true gradient of projected Bellman error

On/Off-Policy	Algorithm	Table Lookup	Linear	Non-Linear
On-Policy	MC	✓	✓	✓
	TD	✓	✓	✗
	Gradient TD	✓	✓	✓
Off-Policy	MC	✓	✓	✓
	TD	✓	✗	✗
	Gradient TD	✓	✓	✓
On/off	Emphatic TD	✓	✓	✓

Convergence of Control Algorithms

Means you chatter. Like you always getting closer but occasionally stepping away but never shoot off to infinity.

Algorithm	Table Lookup	Linear	Non-Linear
Monte-Carlo Control	✓	(✓)	✗
Sarsa	✓	(✓)	✗
Q-learning	✓	✗	✗
Gradient Q-learning	✓	✓	✗

(✓) = chatters around near-optimal value function

Outline

Batch Reinforcement Learning

- Gradient descent is simple and appealing
- But it is *not* sample efficient
- Batch methods seek to find the best fitting value function
- Given the agent's experience ("training data")

Least Squares Prediction <- one way to find best fit

We want to fit our value func approximator to true value func

- Given value function approximation $\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$
- And *experience* $\mathcal{D}$ consisting of $\langle state, value \rangle$ pairs

Oracle tell us these values

$$\mathcal{D} = \{\langle s_1, v_1^\pi \rangle, \langle s_2, v_2^\pi \rangle, ..., \langle s_T, v_T^\pi \rangle\}$$

what's the best fit value func for this whole dataset?

- Which parameters $\mathbf{w}$ give the *best fitting* value fn $\hat{v}(s, \mathbf{w})$?
- Least squares algorithms find parameter vector $\mathbf{w}$ minimising sum-squared error between $\hat{v}(s_t, \mathbf{w})$ and target values v_t^π,

$$LS(\mathbf{w}) = \sum_{t=1}^{T} (v_t^\pi - \hat{v}(s_t, \mathbf{w}))^2$$
$$= \mathbb{E}_{\mathcal{D}} \left[(v^\pi - \hat{v}(s, \mathbf{w}))^2 \right]$$

But how to solve this least squares?
2 solutions:
– experience replay (need some iterations)
– solve directly (only for the special case of linear func approximation) (if #features is not big)

Stochastic Gradient Descent with Experience Replay

It is easy to find least square solution: Experience Replay

Given experience consisting of $\langle state, value \rangle$ pairs

We store this dataset: $\mathcal{D} = \{\langle s_1, v_1^\pi \rangle, \langle s_2, v_2^\pi \rangle, ..., \langle s_T, v_T^\pi \rangle\}$

Repeat:

1. Sample state, value from experience

$$\langle s, v^\pi \rangle \sim \mathcal{D}$$

2. Apply stochastic gradient descent update

$$\Delta \mathbf{w} = \alpha(v^\pi - \hat{v}(s, \mathbf{w}))\nabla_{\mathbf{w}} \hat{v}(s, \mathbf{w})$$

Converges to least squares solution

Global min:

$$\mathbf{w}^\pi = \underset{\mathbf{w}}{\text{argmin}}\ LS(\mathbf{w})$$

This is like supervised learning. We made a dataset, we randomly sample from this dataset update in the direction of the random samples. (This random avoids correlative.)

Experience Replay in Deep Q-Networks (DQN)

Stabilizes the neural networks because it decorrelates the trajectories (via random)

DQN is off policy.

have a second Q networks

DQN uses experience replay and fixed Q-targets

Basically Q–learning

- Take action a_t according to ϵ-greedy policy

Use a neural network to approximate the Q value.

- Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in replay memory $\mathcal{D}$
- Sample random mini-batch of transitions (s, a, r, s') from $\mathcal{D}$

Like 64 random samples from our reply storage/memory

- Compute Q-learning targets w.r.t. old, fixed parameters w^-
- Optimise MSE between Q-network and Q-learning targets

Q–learning prediction of action–value func

Targets are something we plug in to replace oracle

$$\mathcal{L}_i(w_i) = \mathbb{E}_{s,a,r,s' \sim \mathcal{D}_i}\left[\left(r + \gamma \max_{a'} Q(s', a'; w_i^-) - Q(s, a; w_i)\right)^2\right]$$

We freeze the old Q network and we use our targets, basically we bootstrap towards our frozen targets. We don't bootstrap to our least freshest targets, we bootstrap towards our targets from some steps ago and that also stabilized this process. This make it just like supervised learning.

- Using variant of stochastic gradient descent

Sarsa, naive Q–learning, TD may blow up with neural networks
DQN is stable with neural networks. Two reasons are experience replay and fixe Q–targets.

After like 1000 steps we switch around we make our old network equal to our new network and we re–iterate this process. We never bootstrap to the thing we are updating currently bc that can be unstable.

Targets are what we use to replace oracle.

$$LS(\mathbf{w}) = \sum_{t=1}^{T} (v_t^\pi - \hat{v}(s_t, \mathbf{w}))^2$$

$$= \mathbb{E}_{\mathcal{D}} \left[(v^\pi - \hat{v}(s, \mathbf{w}))^2 \right]$$

The danger with TD learning is everytime you update your Q values (w_i here), you also update your target values (w_i−). Bc these two networks are same. Everything you change you parameters (w_i) a little bit, you are changing the targets to which you're moving them. And for non−linear approximation func that can actually spiral out of control.

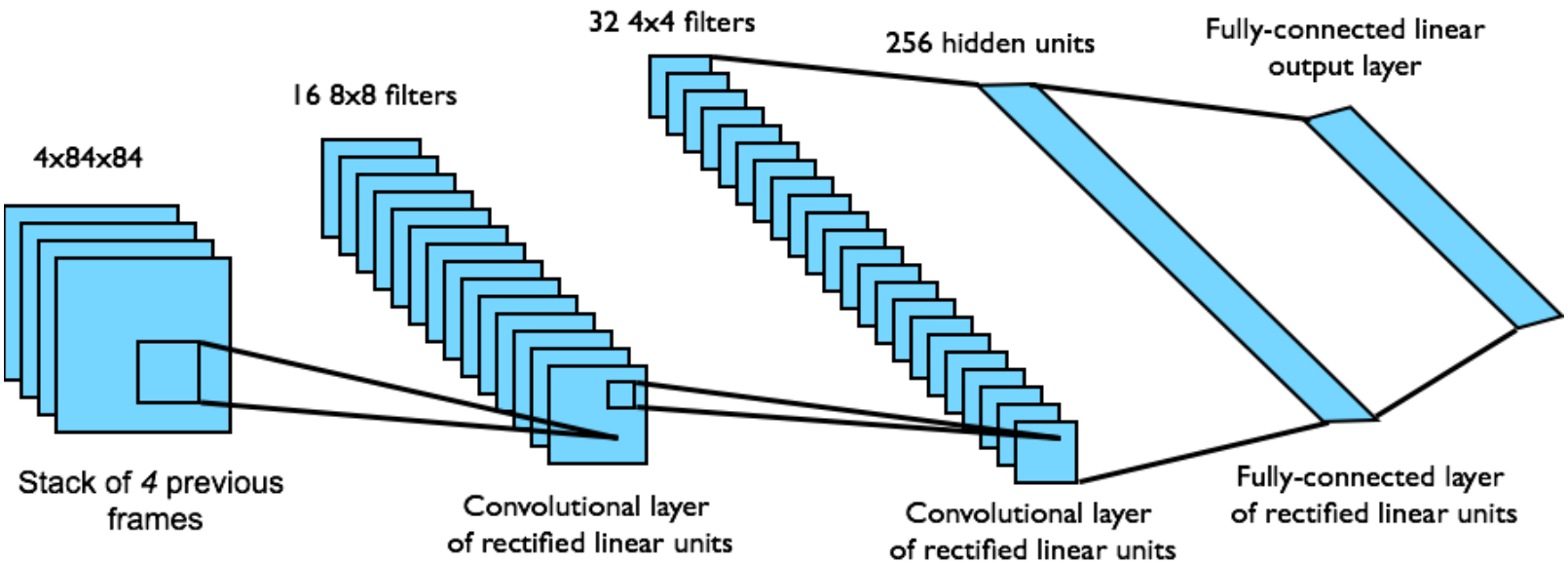
To avoid this, here we freeze the target w_i− to be some old values.

$$\mathcal{L}_i(w_i) = \mathbb{E}_{s,a,r,s' \sim \mathcal{D}_i} \left[\left(r + \gamma \max_{a'} Q(s', a'; w_i^-) - Q(s, a; w_i) \right)^2 \right]$$

Each iteration I, we optimize the loss function L_i(w_i):
We update our latest parameter w_i. We remember some old parameters w_i− (like 1000 steps ago). We use some old parameters, like 1000 steps ago, to generate our target values (Q(w_i−)). The oracle now is substituted by using old parameters, we don't use our freshest most recent parameters to be plugged in to replace oracle. We use some fixed old values, the reason of we want to fix it is bc it gives us a more stable update.

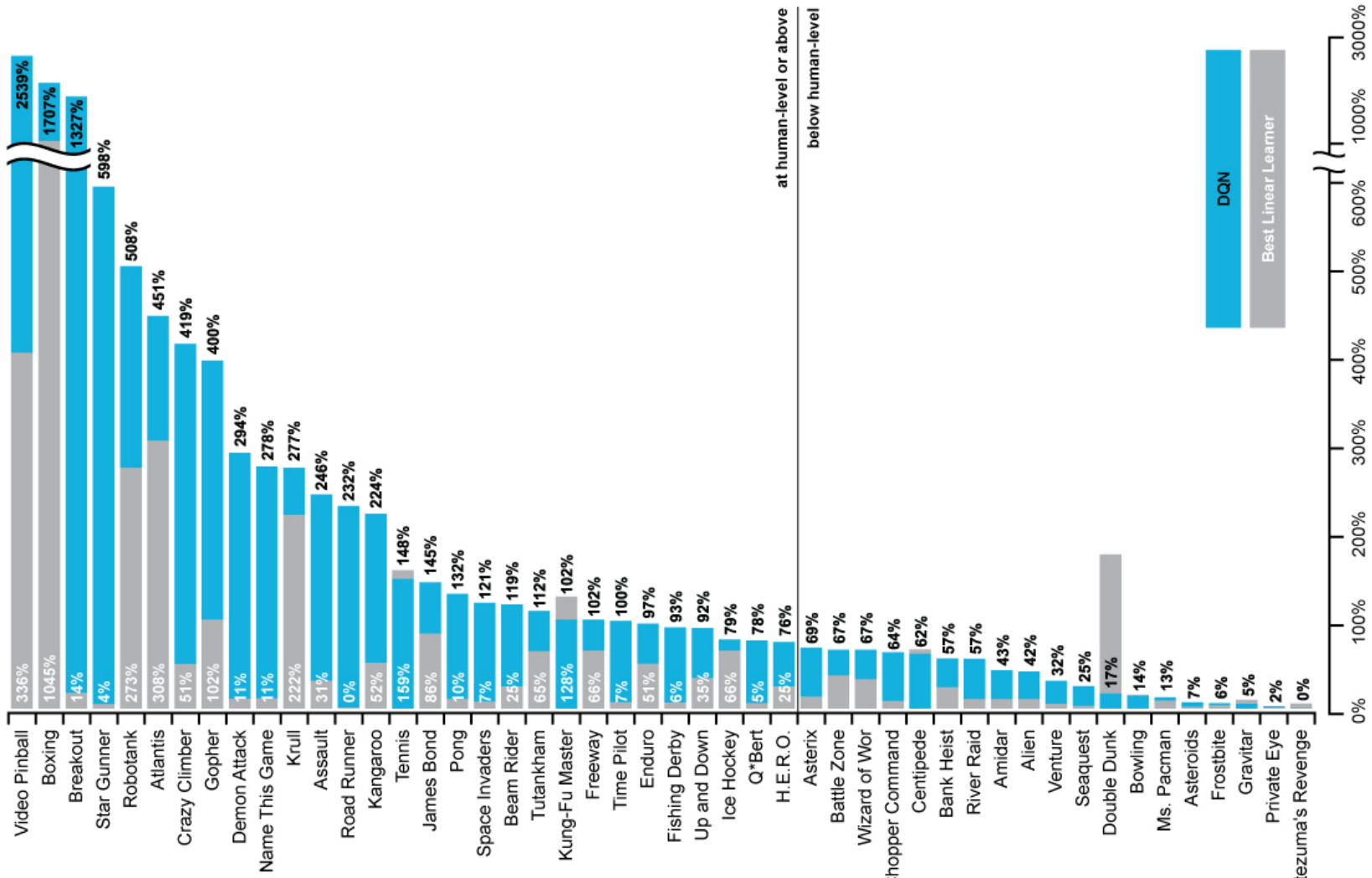
DQN in Atari

- End-to-end learning of values $Q(s, a)$ from pixels s
- Input state s is stack of raw pixels from last 4 frames
- Output is $Q(s, a)$ for 18 joystick/button positions
- Reward is change in score for that step



Network architecture and hyperparameters fixed across all games

DQN Results in Atari



How much does DQN help?

	Replay Fixed-Q	Replay Q-learning	No replay Fixed-Q	No replay Q-learning
Breakout	316.81	240.73	10.16	3.17
Enduro	1006.3	831.25	141.89	29.1
River Raid	7446.62	4102.81	2867.66	1453.02
Seaquest	2894.4	822.55	1003	275.81
Space Invaders	1088.94	826.33	373.22	301.99

Linear Least Squares Prediction

- Experience replay finds least squares solution
- But it may take many iterations
- Using *linear* value function approximation $\hat{v}(s, \mathbf{w}) = \mathbf{x}(s)^\top \mathbf{w}$
- We can solve the least squares solution directly

Linear Least Squares Prediction (2)

- At minimum of $LS(\mathbf{w})$, the expected update must be zero

Is Objective func here

Expected update: $\mathbb{E}_{\mathcal{D}}[\Delta \mathbf{w}] = 0$

If not 0, means we haven't reached the LS solution yet.

Expanding: stepsize*error*gradient

$$\alpha \sum_{t=1}^{T} \mathbf{x}(s_t)(v_t^{\pi} - \mathbf{x}(s_t)^{\top}\mathbf{w}) = 0$$

Step–size

Feature vector

Oracle: we can plug in estimator later (MC, TD)

Linear func approximator

$$\sum_{t=1}^{T} \mathbf{x}(s_t) v_t^{\pi} = \sum_{t=1}^{T} \mathbf{x}(s_t)\mathbf{x}(s_t)^{\top}\mathbf{w}$$

$$\mathbf{w} = \left(\sum_{t=1}^{T} \mathbf{x}(s_t)\mathbf{x}(s_t)^{\top}\right)^{-1} \sum_{t=1}^{T} \mathbf{x}(s_t) v_t^{\pi}$$

- For N features, direct solution time is $O(N^3)$

It doesn't depend on number of states.

- Incremental solution time is $O(N^2)$ using Shermann-Morrison

You can go straight to the solution of your least square with your linear func approimator, find the best weights. Sometimes it's better (cheaper) than experience replay if #features is small. But ER would faster if #features is large.

Linear Least Squares Prediction Algorithms

- We do not know true values v_t^π Oracle
- In practice, our "training data" must use noisy or biased samples of v_t^π

Plug in to replace oracle:

LSMC	Least Squares Monte-Carlo uses return $v_t^\pi \approx G_t$
LSTD	Least Squares Temporal-Difference uses TD target $v_t^\pi \approx R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w})$
LSTD(λ)	Least Squares TD(λ) uses λ-return $v_t^\pi \approx G_t^\lambda$

- In each case solve directly for fixed point of MC / TD / TD(λ)

Linear Least Squares Prediction Algorithms (2)

Making expected updates to 0 and solve the equation:

LSMC

$$0 = \sum_{t=1}^{T} \alpha(G_t - \hat{v}(S_t, \mathbf{w}))\mathbf{x}(S_t)$$

$$\mathbf{w} = \left(\sum_{t=1}^{T} \mathbf{x}(S_t)\mathbf{x}(S_t)^\top\right)^{-1} \sum_{t=1}^{T} \mathbf{x}(S_t) G_t$$

LSTD

$$0 = \sum_{t=1}^{T} \alpha(R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w}))\mathbf{x}(S_t)$$

$$\mathbf{w} = \left(\sum_{t=1}^{T} \mathbf{x}(S_t)(\mathbf{x}(S_t) - \gamma \mathbf{x}(S_{t+1}))^\top\right)^{-1} \sum_{t=1}^{T} \mathbf{x}(S_t) R_{t+1}$$

LSTD(λ)

$$0 = \sum_{t=1}^{T} \alpha \delta_t E_t$$

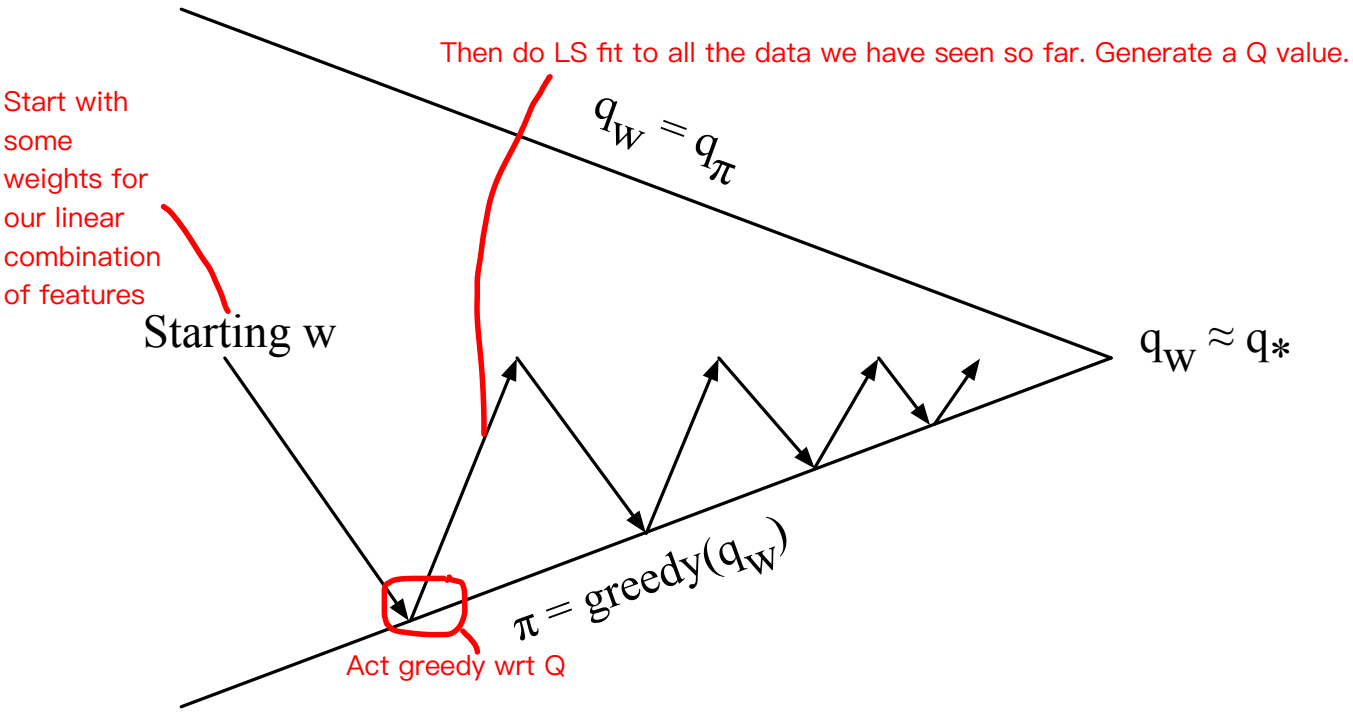
$$\mathbf{w} = \left(\sum_{t=1}^{T} E_t(\mathbf{x}(S_t) - \gamma \mathbf{x}(S_{t+1}))^\top\right)^{-1} \sum_{t=1}^{T} E_t R_{t+1}$$

Convergence of Linear Least Squares Prediction Algorithms

These have better convergence properties than incremental ones.
(But cannot apply to non–linear func approximators)
Converge on both on and off policy. So they have less issues compared with TD which brow up for off–policy.

On/Off-Policy	Algorithm	Table Lookup	Linear	Non-Linear
On-Policy	MC	✓	✓	✓
	LSMC	✓	✓	-
	TD	✓	✓	✗
	LSTD	✓	✓	-
Off-Policy	MC	✓	✓	✓
	LSMC	✓	✓	-
	TD	✓	✗	✗
	LSTD	✓	✓	-

Least Squares Policy Iteration LSPI



Policy evaluation Policy evaluation by least squares Q-learning

Policy improvement Greedy policy improvement

Least Squares Action-Value Function Approximation

- Approximate action-value function $q_\pi(s, a)$
- using linear combination of features $\mathbf{x}(s, a)$

$$\hat{q}(s, a, \mathbf{w}) = \mathbf{x}(s, a)^\top \mathbf{w} \approx q_\pi(s, a)$$

- Minimise least squares error between $\hat{q}(s, a, \mathbf{w})$ and $q_\pi(s, a)$
- from experience generated using policy π
- consisting of $\langle (state, action), value \rangle$ pairs

$$\mathcal{D} = \{\langle (s_1, a_1), v_1^\pi \rangle, \langle (s_2, a_2), v_2^\pi \rangle, ..., \langle (s_T, a_T), v_T^\pi \rangle\}$$

Least Squares Control

- For policy evaluation, we want to efficiently use all experience
- For control, we also want to improve the policy
- This experience is generated from many policies
- So to evaluate $q_\pi(S, A)$ we must learn off-policy
- We use the same idea as Q-learning:
  - Use experience generated by old policy
    $S_t, A_t, R_{t+1}, S_{t+1} \sim \pi_{old}$
  - Consider alternative successor action $A' = \pi_{new}(S_{t+1})$
  - Update $\hat{q}(S_t, A_t, \mathbf{w})$ towards value of alternative action
    $R_{t+1} + \gamma \hat{q}(S_{t+1}, A', \mathbf{w}))$

Least Squares Q-Learning

- Consider the following linear Q-learning update

$$\delta = R_{t+1} + \gamma \hat{q}(S_{t+1}, \pi(S_{t+1}), \mathbf{w}) - \hat{q}(S_t, A_t, \mathbf{w})$$
$$\Delta \mathbf{w} = \alpha \delta \mathbf{x}(S_t, A_t)$$

- LSTDQ algorithm: solve for total update = zero

$$0 = \sum_{t=1}^{T} \alpha(R_{t+1} + \gamma \hat{q}(S_{t+1}, \pi(S_{t+1}), \mathbf{w}) - \hat{q}(S_t, A_t, \mathbf{w}))\mathbf{x}(S_t, A_t)$$

$$\mathbf{w} = \left(\sum_{t=1}^{T} \mathbf{x}(S_t, A_t)(\mathbf{x}(S_t, A_t) - \gamma \mathbf{x}(S_{t+1}, \pi(S_{t+1})))^\top \right)^{-1} \sum_{t=1}^{T} \mathbf{x}(S_t, A_t) R_{t+1}$$

Least Squares Policy Iteration Algorithm

- The following pseudocode uses LSTDQ for policy evaluation
- It repeatedly re-evaluates experience $\mathcal{D}$ with different policies

function LSPI-TD($\mathcal{D}, \pi_0$)
  $\pi' \leftarrow \pi_0$
  repeat
    $\pi \leftarrow \pi'$
    $Q \leftarrow$ **LSTDQ**($\pi, \mathcal{D}$)
    for all $s \in \mathcal{S}$ **do**
      $\pi'(s) \leftarrow \underset{a \in \mathcal{A}}{\text{argmax}}\ Q(s, a)$
    end for
  until $(\pi \approx \pi')$
  return π
end function

Convergence of Control Algorithms

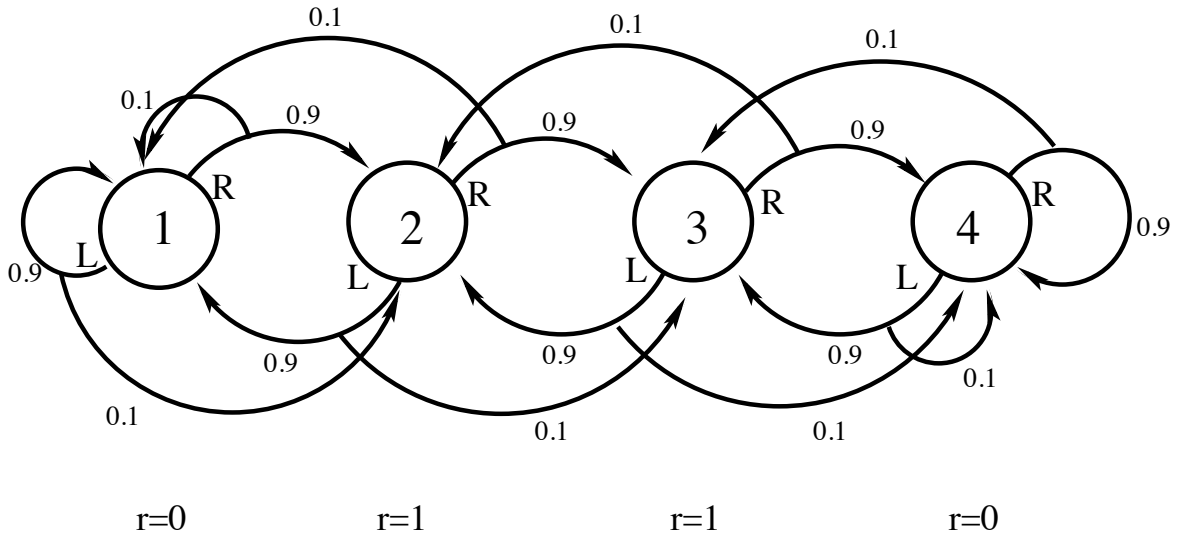
Algorithm	Table Lookup	Linear	Non-Linear
Monte-Carlo Control	✓	(✓)	✗
Sarsa	✓	(✓)	✗
Q-learning	✓	✗	✗
LSPI	✓	(✓)	-

(✓) = chatters around near-optimal value function

When you move to control you still have some issues like chatter. So this not solve all convergence issues.

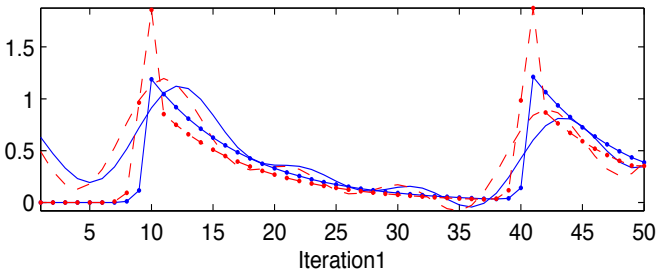
Chain Walk Example

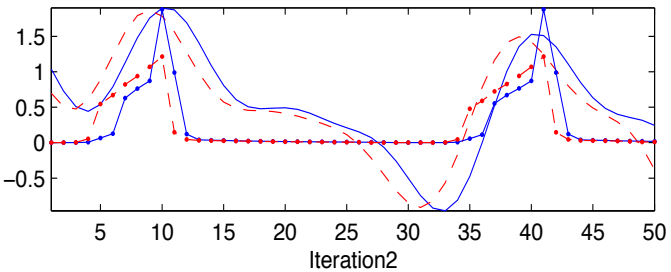
4 state version:

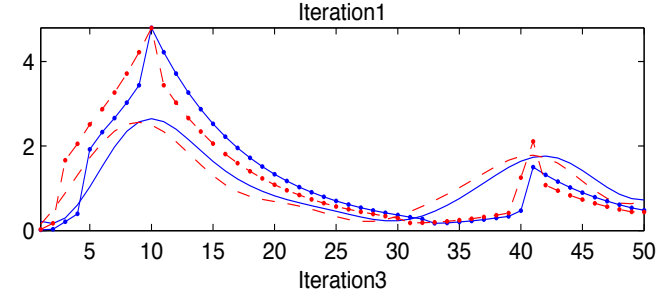


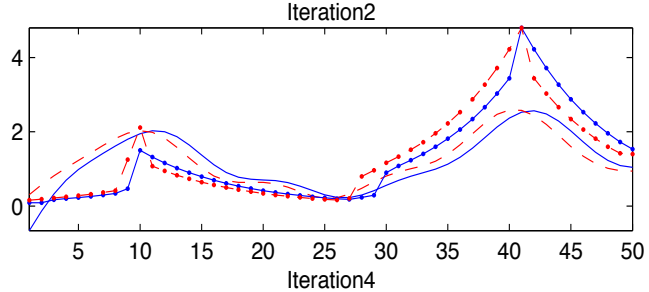
- Consider the 50 state version of this problem
- Reward $+1$ in states 10 and 41, 0 elsewhere
- Optimal policy: R (1-9), L (10-25), R (26-41), L (42, 50)
- Features: 10 evenly spaced Gaussians ($\sigma = 4$) for each action
- Experience: 10,000 steps from random walk policy

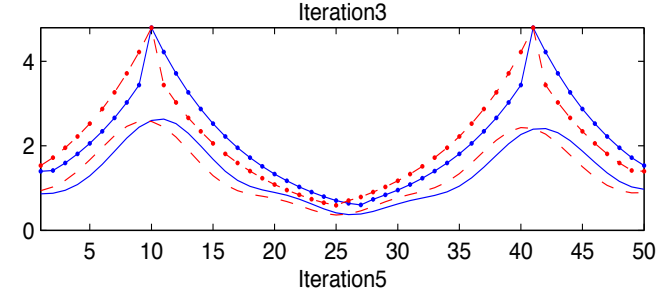
LSPI in Chain Walk: Action-Value Function

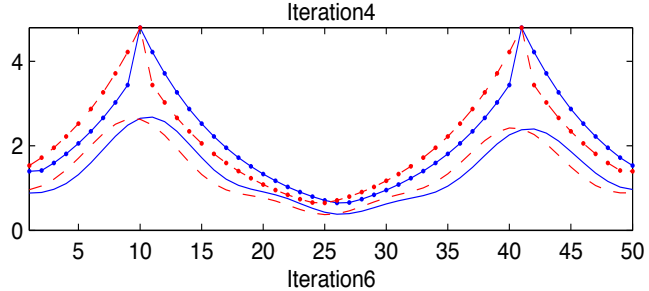


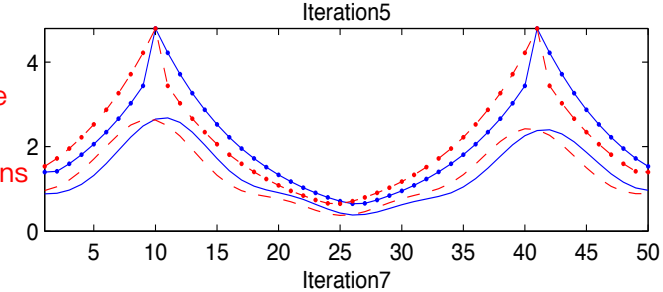








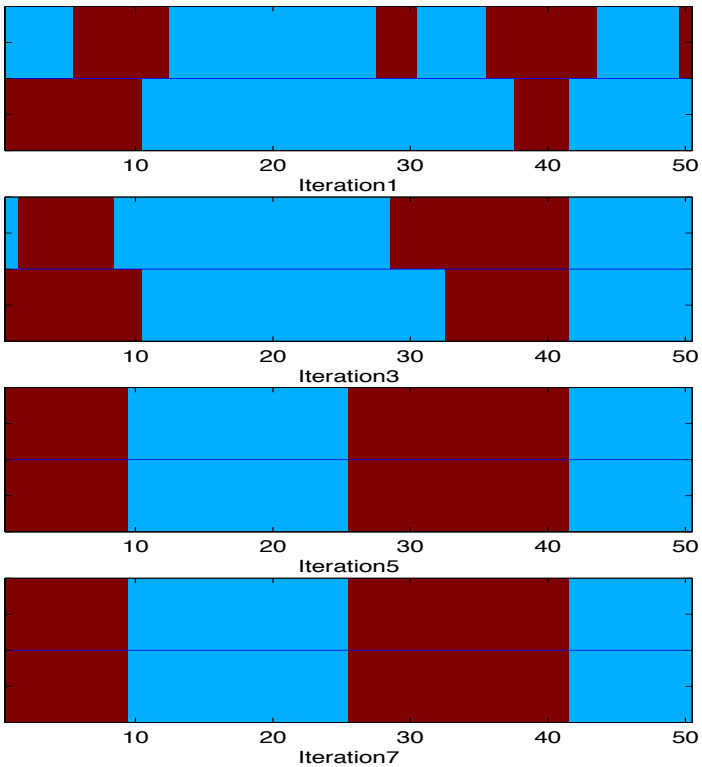




LSPI learns the perfect shape after 7 iterations

Blue: action go left
Red: action go right
Dash: true value func
Solid: approximate value func

LSPI in Chain Walk: Policy



Learned best actions: red is go right; blue is go left.

Questions?

Next time: policy based methods for scaling up using func approximation